Table of Contents


```
%
% Copyright (c) 2015, Mostapha Kalami Heris & Yarpiz (www.yarpiz.com)
% All rights reserved. Please read the "LICENSE" file for license
 terms.
%
% Project Code: YPEA120
% Project Title: Non-dominated Sorting Genetic Algorithm II (NSGA-II)
% Publisher: Yarpiz (www.yarpiz.com)
%
% Developer: Mostapha Kalami Heris (Member of Yarpiz Team)
%
% Cite as:
% Mostapha Kalami Heris, NSGA-II in MATLAB (URL: https://
yarpiz.com/56/ypea120-nsga2), Yarpiz, 2015.
%
% Contact Info: sm.kalami@gmail.com, info@yarpiz.com
%

clc;
clear;
close all;
```

Problem Definition

```
CostFunction = @(x) MOP4(x);      % Cost Function

nVar = 3;             % Number of Decision Variables

VarSize = [1 nVar];   % Size of Decision Variables Matrix

VarMin = -5;          % Lower Bound of Variables
VarMax = 5;          % Upper Bound of Variables

% Number of Objective Functions
nObj = numel(CostFunction(unifrnd(VarMin, VarMax, VarSize)));
```

Testing

```
% Independent variable
popVals = [40 80 120 160 200];        % Population Size
kungs = 1;  % whether to use naive (0) or kung's (1) method
```

```
numTrials = 200;    % trials per scenario

% Statistics
executiontime = zeros(2, width(popVals), numTrials);
avgTime = zeros(2, width(popVals));

% algorithm parameters
MaxIt = 10;      % Maximum Number of Iterations
pCrossover = 0.7;                         % Crossover Percentage
pMutation = 0.4;                          % Mutation Percentage
mu = 0.02;                   % Mutation Rate
sigma = 0.1*(VarMax-VarMin);  % Mutation Step Size
for kungs = 0:1
    for pIdx = 1:width(popVals)
        nPop = popVals(pIdx);        % Population Size
        nCrossover = 2*round(pCrossover*nPop/2);  % Number of Parents
 (Offsprings)
        nMutation = round(pMutation*nPop);        % Number of Mutants
        for t = 1:numTrials
            % Initialization
            empty_individual.Position = [];
            empty_individual.Cost = [];
            empty_individual.Rank = [];
            empty_individual.DominationSet = [];
            empty_individual.DominatedCount = [];
            empty_individual.CrowdingDistance = [];
            pop = repmat(empty_individual, nPop, 1);
            for i = 1:nPop
                pop(i).Position = unifrnd(VarMin, VarMax, VarSize);
                pop(i).Cost = CostFunction(pop(i).Position);
            end

            % Non-Dominated Sorting
            [pop, F] = NonDominatedSorting(pop);

            % Calculate Crowding Distance
            pop = CalcCrowdingDistance(pop, F);

            % Sort Population
            [pop, F] = SortPopulation(pop);


            % NSGA-II Main Loop
            tic;
            for it = 1:MaxIt
                % Crossover
                popc = repmat(empty_individual, nCrossover/2, 2);
                for k = 1:nCrossover/2
                    i1 = randi([1 nPop]);
                    p1 = pop(i1);
                    i2 = randi([1 nPop]);
                    p2 = pop(i2);
                    [popc(k, 1).Position, popc(k, 2).Position] =
 Crossover(p1.Position, p2.Position);
```

```
                    popc(k, 1).Cost = CostFunction(popc(k,
1).Position);
                    popc(k, 2).Cost = CostFunction(popc(k,
2).Position);
                end
                popc = popc(:);
                % Mutation
                popm = repmat(empty_individual, nMutation, 1);
                for k = 1:nMutation
                    i = randi([1 nPop]);
                    p = pop(i);
                    popm(k).Position = Mutate(p.Position, mu, sigma);
                    popm(k).Cost = CostFunction(popm(k).Position);
                end

                % Merge
                pop = [pop
                     popc
                     popm]; %#ok

                 % algorithm
                if kungs
                    % sort by 1st objective
                    carr = [pop.Cost];
                    [~, idx] = sort(carr(1,:), 2, 'ascend');
                    pop = pop(idx);

                    % deterine fronts
                    idx = 1:height(pop);
                    fr = 1;
                    while size(idx) > 0
                        [~,F{fr}] = Front(pop(idx), idx);
                        for fi = F{fr}
                            pop(fi).Rank = fr;
                        end
                        idx = setdiff(idx, F{fr});
                        fr = fr + 1;
                    end

                    % truncate
                    pop = CalcCrowdingDistance(pop, F);
                    [pop, F] = SortPopulation(pop);
                    pop = pop(1:nPop);

                    % sort once more
                    [pop, F] = SortPopulation(pop);

                    % store F1
                    F1 = pop(F{1});
                else
                    % Non-Dominated Sorting
                    [pop, F] = NonDominatedSorting(pop);

                    % Calculate Crowding Distance
```

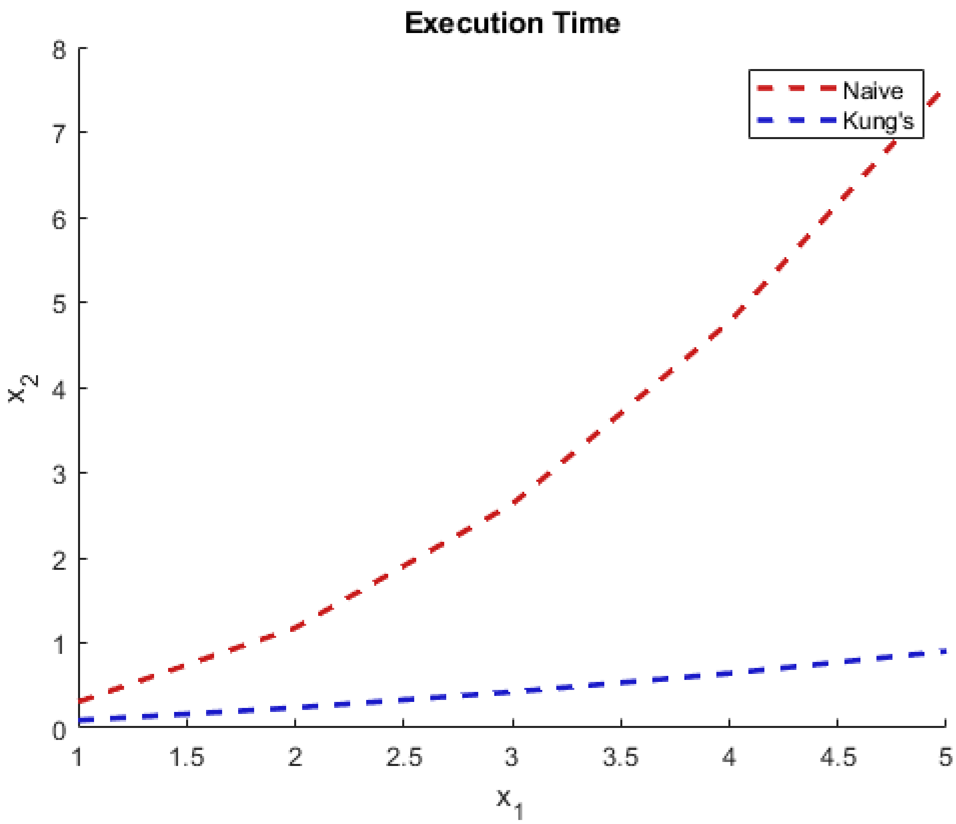
```
                    pop = CalcCrowdingDistance(pop, F);

                    % Sort Population
                    pop = SortPopulation(pop);

                    % Truncate
                    pop = pop(1:nPop);

                    % Non-Dominated Sorting
                    [pop, F] = NonDominatedSorting(pop);

                    % Calculate Crowding Distance
                    pop = CalcCrowdingDistance(pop, F);

                    % Sort Population
                    [pop, F] = SortPopulation(pop);

                    % Store F1
                    F1 = pop(F{1});
                end

                % Show Iteration Information
                %disp(['Iteration ' num2str(it) ': Number of F1
 Members = ' num2str(numel(F1))]);

                % Plot F1 Costs
                %figure(1);
                %PlotCosts(F1);
                %pause(0.01);
            end
            executiontime(kungs+1, pIdx, t) = toc;
        end
        disp(['Completed nPop=' num2str(nPop) ' with kungs='
 num2str(kungs)]);
    end
end
```

```
Completed nPop=40 with kungs=0
Completed nPop=80 with kungs=0
Completed nPop=120 with kungs=0
Completed nPop=160 with kungs=0
Completed nPop=200 with kungs=0
Completed nPop=40 with kungs=1
Completed nPop=80 with kungs=1
Completed nPop=120 with kungs=1
Completed nPop=160 with kungs=1
Completed nPop=200 with kungs=1
```

Results

```
for p = 1:width(popVals)
    avgTime(1,p) = mean(executiontime(1,p,:));
    avgTime(2,p) = mean(executiontime(2,p,:));
```

```
end
figure();
hold on;
title('Execution Time');
xlabel('x_1');
ylabel('x_2');
ax = gca;
ax.XAxisLocation='origin';
ax.YAxisLocation='origin';
x1 = 1:width(popVals);
plot(x1, avgTime(1,:), 'Color',
[.8 .1 .1], 'LineStyle', '--','LineWidth',2,'DisplayName','Naive');
plot(x1, avgTime(2,:), 'Color',
[.1 .1 .8], 'LineStyle', '--','LineWidth',2,'DisplayName',"Kung's");
legend();
```



Comparison

Kung's method scales significantly better than the naive implementation. At a lower population size the difference isn't as significant, but the scaling of the naive implementation fails relatively quickly. Execution time thus skyrockets, while Kung's method stays relatively the same. This aligns with the statement that Naive is O(dn^2) and Kung's is O(n log(n)). Kung's, however, might use more memory. Since matlab is a pass-by-value language, every recursion of Kung's gets its own copy of the data. This probably ends up being more than the single copy of pop that the naive implementation creates. However, I have no way of accurately measuring this, so I can only theorize. Regardless, the extra memory is absolutely worth the increase in execution speeds, which becomes roughly tenfold for the population size of 200.

Published with MATLAB® R2021a